Interactive Data Representation Migration: Exploiting Program Dependence to Aid Program Transformation

Krishna Narasimhan, Christoph Reichenbach, Julia Lawall

Interactive Data Representation Migration

Exploiting Program Dependence to Aid Program Transformation

Krishna Narasimhan
Goethe University, Frankfurt, Germany
krishna.nm86@gmail.com

Christoph Reichenbach
R^2 Software & Systeme UG, Germany
creichen@gmail.com

Julia Lawall
Sorbonne Universités/UPMC/Inria/LIP6, France
Julia.Lawall@lip6.fr

Abstract

Data representation migration is a program transformation that involves changing the type of a particular data structure, and then updating all of the operations that somehow depend on that data structure according to the new type. Changing the data representation can provide benefits such as improving efficiency and improving the quality of the computed results. Performing such a transformation is challenging, because it requires applying data-type specific changes to code fragments that may be widely scattered throughout the source code, connected by dataflow dependencies. Refactoring systems are typically sensitive to dataflow dependencies, but are not programmable with respect to the features of particular data types. Existing program transformation languages provide the needed flexibility, but do not concisely support reasoning about dataflow dependencies.

To address the needs of data representation migration, we propose a new approach to program transformation that relies on a notion of semantic dependency: every transformation step propagates the transformation process onward to code that somehow depends on the transformed code. Our approach provides a declarative transformation-specification language, for expressing type-specific transformation rules. We further provide scoped rules, a mechanism for guiding rule application, and tags, a device for simple program analysis within our framework, to enable more powerful program transformations.

We have implemented a prototype transformation system based on these ideas for C and C++ code and evaluate it against three example specifications, including vectorization, transformation of integers to big integers, and transformation of array-of-structures data types to structure-of-arrays format. Our evaluation shows that our approach can improve program performance and the precision of the computed results, and that it scales to programs of up to 3700 lines.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques

Keywords Static Analysis, Program Transformation, DSL

PEPM '17 January 16–17, 2017, Paris, France

ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00
DOI: http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

1. Introduction

Evolution at the source code level is inevitable in any real-world software system [13], to improve security, maintainability, and performance, and to address new needs. A frequently useful type of evolution is a migration between data representations. For example, converting integers to 'bigints' enables computations on larger values, converting an array of structures to a structure of arrays can improve locality, and vectorization at the source-code level can better exploit the capabilities of a machine that provides vectorized instructions. Such evolution may require disparate but interconnected transformations all over a program.

Consider the problem of vectorization. Vectorization enables software to simultaneously perform various arithmetic operations on adjacent array elements, rather than individually on scalar values. The Vc API [12] provides a collection of functions and datatypes that ensure explicit vectorization of C++ code at the source level. Unlike compiler-level vectorization, which may or may not succeed, source-level vectorization guarantees vectorized execution [11]. The Vc API provides custom types, such as `float_v` representing a `float` vector, which has the capability to hold a fixed-size sequence of `float`s. Vectorization with the Vc API thus requires users to change type declarations, operations that use values of these types, custom data structures that hold the results of these operations, functions that receive or return the new types of values, and so on. Performing multiple such scattered changes quickly becomes tedious and error-prone. We therefore argue for the need for tool support. To address the needs of various kinds of data types, such a tool should be easily extensible.

Existing tools for automated program transformation can support such migration to some degree: users can write a specification for e.g. Stratego [3] or Coccinelle [17] and ask these tools to apply the specification to the entire program. Unfortunately, this process does not consider the issue that a user may want a transformation to be applied selectively, to only specific instances of a data type, for semantic or performance reasons. For example, consider migrating from `int` to a `bignum` type in order to scale an arithmetic module to support larger numbers. In a C, C++, or Java program, this is not a change that we would want to automatically apply to all `int` variables in the entire program. Instead, we will typically want to be able to select a subset of the integer variables, and have the system automatically transform all of these dependencies. Refactoring systems like the Eclipse IDE [6] provide the option of choosing a starting point to apply the transformation and then the system automatically applies the transformation to all the uses of the transformed code. For example, renaming a variable in one location using the Eclipse refactoring system will rename all the other uses of the variable. However, refactoring systems come with a predefined set of simple transformations and are not customizable like transformation languages. What we need for data representation migra-

tion is a combination of refactoring-like user-driven selection, plus dependency tracking, with the customizability of program transformation languages.

We propose a tool, **DMF** (Data Migration Framework), that supports semi-automated migrations of data representations. **DMF** provides a rule-specification language, coupled with a transformation engine that applies the rules to C or C++ code based on a starting point chosen by the user and dependency-driven search across the code base. Our language provides four main features in accordance with our analysis above: 1) *Search* across dependencies of transformed terms to identify terms requiring transformation, 2) *Scoped rules* that help direct the application of rules only to particular contexts, such as loop bodies or structure definitions, 3) *Tags* that enable propagating information gathered from simple forms of program analysis on AST fragments, and 4) *User guidance* including the choice of where in the code to start the transformation and when to roll back a series of transformation steps. **DMF** is implemented within the Eclipse framework [6], and relies on the CDT C/C++ program manipulation plugin [7].

The main contributions of our work are as follows:

- We study a motivating example from a real problem of migrating from scalar to vector representations at the source level.
- Based on the example, we analyze and classify the kinds of code transformations required to perform data representation migrations.
- We propose a transformation language addressing these requirements, and informally describe its semantics.
- We apply our transformation system on existing use cases and discuss the performance and precision of the resulting code.
- We show that the running time of **DMF** on a non-trivial program is reasonable in spite of the approach's reliance on user feedback and multi-directional search along the program's abstract syntax tree and the program dependency graph.
- We also show the generality of our approach by applying an existing specification in our language to open source code that is randomly selected from GitHub.

The rest of this paper is organized as follows. Section 2 presents a motivating example, in terms of code fragments that illustrate the transformation to be performed. Section 3 presents the main concepts of our transformation language. Section 4 gives an overview of the language semantics. Section 5 evaluates our approach on several case studies. Section 6 studies the code resulting from transforming the case studies and analyzes its performance and precision. Section 7 discusses the scalability of our approach on a non-trivial transformation and the application of a previously created specification in our language to code from randomly selected software projects in GitHub. Section 8 presents related work and Section 9 concludes.

2. Motivating Example

We begin with the example of vectorization to illustrate the challenges in performing data representation migration. The Vc library provides vector versions of primitive types and corresponding operations to allow developers to write portable versions of vectorized code [12]. Figure 1 shows a program that converts an array of Cartesian coordinates, represented using floats, into polar coordinates (left) and a program that does the same, but on an array of float vectors (right). In the latter, the vector type float_v represents a machine vector of float variables; its size is hardware-dependent. As we see in the line below the comment *vectorized conditional update* in the right hand side code, Vc can vectorize not only arithmetic operations and updates, but also comparisons and conditional

```
1  struct CartesianCoordinate {
2      float x;
3      float y;
4  };
5  CartesianCoordinate input[1000];
6
7  struct PolarCoordinate {
8      float r;
9      float phi;
10 };
11 PolarCoordinate output[1000];
12
13 void fn() {
14   for (int i = 0; i < 1000; ++i) {
15     float x = input[i].x;
16     float y = input[i].y;
17     output[i].r =
18       std::sqrt((x * x) + (y * y));
19     output[i].phi =
20       std::atan2(y, x)
21       * 57.29578018188476f;
22     if (output[i].phi < 0.f) {
23       output[i].phi += 360.f;
24     }}}
```

```
struct CartesianCoordinate_v {
    float_v xv;
    float_v yv;
};
CartesianCoordinate_v
  input_v[1000 / float_v::size];

struct PolarCoordinate_v {
    float_v rv;
    float_v phiv;
};
PolarCoordinate_v
  output_v[1000 / float_v::size];

void fn() {
  for (int i = 0; i <
  1000 / float_v::size; ++i) {
    float_v x = input_v[i].xv;
    float_v y = input_v[i].yv;
    output_v[i].rv =
      vc::sqrt((x * x) + (y * y));
    output_v[i].phiv =
      vc::atan2(y, x) *
      57.29578018188476f;
    // vectorized conditional update:
    output_v[i].phiv
      (output_v[i].phiv < 0.f)
        += 360.f;
  }}
```

Figure 1. Vectorization using the Vc library

updates, using C++ function call syntax. This example is inspired by a transformation tutorial found on the Vc website [10].

The main challenges in vectorizing this code are as follows:

- The developer must identify and transform syntactic patterns, such as the transformation from the call to **std::sqrt** to the call to **vc::sqrt** .
- The Vc documentation describes migration from an array of structures to another array of structures with its fields vectorized. However, the vectorization is only likely to improve performance when all fields have types that have the same in-memory sizes. For simplicity, we require that all field types be the same. The developer could thus start by analyzing the definition of struct CartesianCoordinate, which is the type of **input**, and observe that it contains only floats. The developer can then create a new vectorized type CartesianCoordinate_v, rename **input** to **input_v**, and update CartesianCoordinate_v's float fields to use the float_v type.
- Vectorizing the elements of **input** changes the array size. The developer must thus collect information about the type of the structure fields, float in our example, and use the size of its Vc counterpart, float_v, to compute the new size. A similar change is needed on the limit of the **for** loop, based on the type of the array element expressions inside the loop.
- The developer must transform all the uses of **input** to **input_v**.

To automate these steps, we need a tool with the following abilities:

1. Propagating the need for transformation across variable def-use dependencies. For example, transforming **CartesianCoordinate input[1000];** must trigger processing of all uses of **input**.
2. Transforming and analyzing code sub-fragments. For example, the transformation must consistently apply the float-to-float_v transformation to all declarations in the bodies of relevant type definitions.
3. Propagating information from one transformation step to another. For example, changing the size of **input** requires knowing that the fields of CartesianCoordinate_v have type float_v,

as the new size depends on the size of the float vector in the current hardware implementation.

4. Rolling back the transformation, if the transformation process detects inconsistencies.

3. Language

We now present an overview of our transformation language and highlight how it meets the needs identified in Section 2. We present its semantics in Section 4.

3.1 Rules

A transformation specification in our language consists of a sequence of *top-level* rules which can in turn contain invocations of *scoped* rules.

3.1.1 Top-Level Rules

A *top-level* rules allows users to specify pattern matching and transformation and has the form

Category *pattern* ==> *transformation*
[**notransform** *term_list*]
[**where** *condition*]

Category is a syntactic category, such as **expression**, **statement**, **declaration**, or **decldefinition**. The **decldefinition** category, illustrated in Section 3.1.3, indicates a pattern that matches a variable declaration and the definition of its type at once. *Pattern* is a pattern that has the form of a term in the specified syntactic category and may contain *metavariables*, which match arbitrary subterms. The name of a metavariable begins and ends with $. *Transformation* is another pattern, which describes the generated code. To transform code, we match the code against *pattern*, bind all metavariables, and substitute them in *transformation*. In our language, application of a transformation rule triggers the processing of terms that somehow depend on the transformed term. The optional **notransform** clause contains subterms on which such triggering should not occur. The optional **where** clause puts some constraints on the possible values of the metavariables. Constraints currently relate to the type of an expression-typed metavariable, where the type is obtained using the C++-like operator **decltype** [5], and equality checks on the structure of the code bound to the metavariables. We envision that this list of supported **where** clauses can be extended.

The GMP library [9] from GNU provides an API for performing arbitrary precision arithmetic. As an example of a transformation specification, the following top-level rules transform an integer multiplication to its big integer counterpart in the GMP library:

```
expression $a$ = $b$ * $c$ ==> mpz_mul_si ($a$, $b$, $c$)
notransform: $c$
where decltype($c$) == int

expression $a$ = $b$ * $c$ ==> mpz_mul_ui ($a$, $b$, $c$)
notransform: $c$
where decltype($c$) == unsigned int
```

In the first rule, the category is **expression**, the pattern is **a = b * c** and the transformation is **mpz_mul_si(a, b, c)**; the second rule is structured similarly. These rules transform a multiplication and assignment into the bignum multiplication functions *mpz_mul_si*, for signed integers, and *mpz_mul_ui*, for unsigned integers. Each generated function call has three arguments. The first two are of type **mpz_t** and the last one is an integer, i.e., an int or unsigned int, as it is in the original code. Both rules thus use **notransform** to indicate that this third argument should not be scheduled for further transformation. The application of top-level rules follows a dependency-based strategy. Specifically, top-level rules trigger on code if either the user selected that code, or if that depends on a code where a transformation was applied.

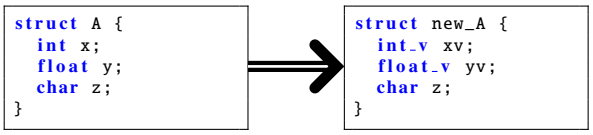

Figure 2. Struct definition before and after applying vc_decl_scope

3.1.2 Scoped Rules

The dependency-based application strategy for top-level rules is effective for following the program's data flow, but in some cases we must transform a region of code exhaustively. For example, in our Vc example, transforming a structure involves transforming int and float declarations into int_v and float_v declarations, exhaustively within the body of a type definition. We support exhaustive transformation of a region of code with *scoped rules*, which are sets of transformation rules that are limited to a particular region of code and applied to every syntactic match in the region.

A scoped rule has the form:

scope *name*[(*parameter_list*)]{{
(**tag** *typename*;)*
transformation_rule+
}}

A simple example of a scoped rule is:

```
scope vc_decl_scope {{
  declaration int $a$; ==> int_v $a$v;
  declaration float $a$; ==> float_v $a$v;
}}
```

This rule replaces all occurrences of an int declaration in the given scope by an int_v declaration and renames the declared variable or field by appending a 'v'; it transforms float declarations analogously. In our motivating example in Figure 1, we would use this scoped rule to change the types and names of all fields in a specific structure.

A scoped rule does not explicitly specify the scope to which it applies. Instead, top-level rules trigger scoped rules explicitly, within their *transformation* specification. The syntax mirrors that of a function call, as illustrated below:

```
declaration struct $s$ { $body$ } ==>
struct new_$s$ { vc_decl_scope($body$) }
```

In this case, the transformed term is constructed by instantiating the metavariable s as indicated by the *pattern*, and by replacing the scoped rule invocation, **vc_decl_scope($body$)**, by the result of applying the rule vc_decl_scope exhaustively within the term that has been bound according to *pattern* to the metavariable $body$.

As an example of the use of the above rule, consider the structure declaration in Figure 2. Applying the above top-level rule binds the metavariable s to the structure name A and the metavariable $body$ to the sequence of field declarations, int x; etc. The transformation component of the rule indicates that the resulting structure should be named new_A, as shown in Figure 2, and the declared fields should be the result of applying the scoped rule vc_decl_scope anywhere it matches within the field declaration list, specifically, to the int and float fields.

3.1.3 Tags

We further allow reasoning over the code to ensure uniformity constraints through scoped rule *tags*. We observed previously that vectorization works best when all of the fields of the transformed structure have the same type. As illustrated by Figure 2, the scoped rule vc_decl_scope currently does not ensure this property. To

```
decldefinition
  struct $structname$ {$body$} $obj_name$[$s$] ==>
  struct $structname$v {vc_struct($body$)}
  $obj_name$v [$s$/vc_struct.vctype::size]
```

Figure 3. Example usage of tags

address this issue, we add a *tag* to the scoped rule. A tag serves as local storage for the processing of a scope. A tag is declared as **tag** *tagname* [=initial value];. If no initial value is supplied, then the tag is initialized to $\perp$. The tag can be initialized once in the processing of a scope, and then subsequent attempts to modify it must provide the same value, effectively implementing universal ($\forall$) quantification. Tags are updated using an **updatetag** clause that can have one of the following forms:

updatetag $tagname := value$
updatetag $tagname := (\ (value : condition,)+\ )$

An **updatetag** clause follows an individual rule inside a scoped rule. It can update a tag with a specific value, or with one of a set of values, whichever satisfies a corresponding condition. The set of conditions do not have to be exhaustive. A default value can be provided by setting the condition to **otherwise**. If there is no default value and the conditions are not exhaustive, then unmatched cases result in no update.

The following example extends the scoped rule from Section 3.1.2 to include a tag:

```
scope vc_struct {{
  tag vctype;
  declaration int $a$; ==> int_v $a$v;
    {{updatetag vctype:=int_v}}
  declaration float $a$; ==> float_v $a$v;
    {{updatetag vctype:=float_v}}
  declaration $T$ $a$; ==> $T$ $a$;
    {{updatetag vctype:=top}}
}}
```

This scoped rule provides the same transformation rules as `vc_decl_scope` above, but on each transformation it also updates the tag `vctype` with the chosen type. Application of the scoped rule only succeeds if all the updates to the tag have the same value; otherwise, the entire transformation process is aborted, reverting the code in its original state. In this example, the first two rules update the tag to the chosen type, which is then used in the transformed code, while the third rule actually performs no transformation, but only gives the tag the value $\top$ (top) to indicate that a type other than int or float has been detected and the scoped rule application should fail. Since we prioritize earlier rules over later rules, this third rule does not match if `$T$` is int or float. The tag thus ensures that the scope does not contain a mixture of ints and floats, or of ints or floats and other types. In particular, our example structure declaration in Figure 2 would incur a failure and a rollback of the whole transformation process.

Tag values can also be accessed by the rule that invokes the scoped rule using the form *scopedrule.tagname*, allowing the former rule to obtain information about the code processed in the scope. In the rule in Figure 3, our language uses the tag value obtained by processing the type definition in specifying the new size of the array-typed variable. If this functionality is used, the scoped rule can be used only once in the given transformation.

In the context of the CartesianCoordinate structure defined on the left side of Figure 1, the scoped rule **vc_struct** will update the tag **vctype** with float_v. Here, **size** is an attribute of vector types in the Vc API, such as float_v and int_v. Then, applying the rule in Figure 3 on the **decldefinition** comprising the type definition of the struct CartesianCoordinate and the declaration **struct CartesianCoordinate input[1000]** will yield a new type, CartesianCoordinate_v, along with its definition, and a new object of that type, **input_v**, as shown on the right side of Figure 1.

The rule in Figure 3 illustrates the use of the syntactic category **decldefinition**, which is useful when the transformation of a variable declaration, in our case the array declaration, requires information based on analysis of the definition of the type, in our case the structure definition.

3.1.4 Scoped Rules with Parameters

A scoped rule that is parameterized can be called by a top-level rule which passes values to these parameters. The scoped rule can in turn use the supplied values to perform checks on the terms processed in the scope or use the values to perform transformations based on these values. As an example, consider the requirement that all of the array indexed elements inside a **for** loop must be indexed by the index of the **for** loop. We can express such a rule using the following scoped rule. A detailed specification of this rule for the general loop scenario is described in Section 5.

```
scope vc_for($index$) {{
  tag loopindex =  $ind$;
  expression $a$[$i$ ].$b$ ==> $a$[$i$ ].$b$
    updatetag loopindex := $i$
}}
```

An example use of such a parameterized scoped rule is as follows:

```
statement
for(int $i$ = 0; $i$ < $limit$; $i$++){$body$}
==>
for(int $i$ = 0; $i$ < $limit$/vc_for.vctype::size; $i$++)
{ vc_for($body$, $i$) }
```

In a scoped rule invocation, the first argument represents the subterm to which the scoped rule should apply, and the formal parameters are bound to the remaining arguments.

3.2 Assessment

Comparing the features of our language to the requirements that we outlined in Section 2, we find that our notion of propagating change across dependencies (which we detail in the next section) supports requirement (1). Scoped rules support requirement (2), while tags support requirement (3), allowing us to pass information from scoped to top-level rules. We have furthermore noted a case where the transformation language's run-time system provides rollback, thus addressing requirement (4), in the context of scoped rules.

4. Semantics

The semantics of our transformation language specifies *where* transformations are performed, *i.e., which* terms are selected for transformation, and *how* the transformation of the selected terms is carried out. The latter is straightforward and typical of rule based approaches in which rules are expressed using concrete syntax: we match a pattern against source code, instantiate the transformation specification according to the match information, and finally schedule the original code to be replaced by the result. Our contribution lies in the choice of where to perform transformation, to meet the needs of data migration. For this we rely on two key notions: propagation over dependencies and user interaction. We start with our program model that supports these features.

4.1 Program Model

The goal of our language is to change how code represents data. Changing a data representation requires changing types, which naturally affects all of the operations that process that data. Changing these operations may in turn affect their other inputs: for example, vectorizing a variable may trigger the vectorization of a binary operator that uses that variable, which in turn may require vectorizing

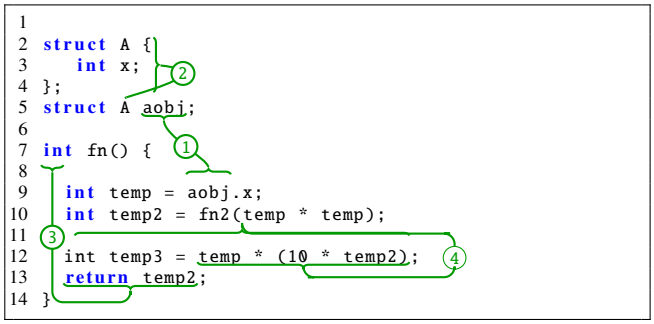

Figure 4. Dependency links

the other argument as well. The change may also affect the representation of the operation's result, thus requiring changes in other operations that depend on this information. Accordingly, reasoning about dependencies between program elements is a key part of the semantics of our language.

We represent the source code as an abstract syntax tree (AST), in which nodes are annotated with *dependency links*. A dependency link is a symmetric connection between two nodes, such that if the system transforms one of the nodes, the other must also be considered for transformation. The exact positioning of the dependency links depends on the source language. Our implementation for C and C++ programs supports the following dependency links:

1. Between a variable's uses and its declaration and definitions.
2. Between a type reference and its definition.
3. Between a function and its callers, i.e.,
   - Between a function's return statements and the call sites.
   - Between a function's actual and formal parameters.
4. Between a node and its immediate parent.

The first three types of links are also found in Program Dependence Graphs [8]. The fourth type is derived from the fact that a change in a term can affect the side effect or result of that term, and thus may require changing any containing term that observes that effect or result, and inversely a change in an operation may require changing the computations whose results are manipulated by that operation. Our approach currently does not take aliases into account, but we envision that it could be extended with an existing alias analysis for C or C++ code [23]

Figure 4 illustrates all four types of links: Link type (1) occurs between the declaration of `aobj` in line 5 and its use in line 9, while type (2) occurs between the definition of struct `A` and its use in line 5. Type (3) connects the function's return statement with its call sites, and type (4) connects the expression `temp * (10 * temp2)` with the declaration and initialization of `temp3`.

4.2 Algorithm

The algorithm, outlined in Figure 5, begins with a node, *SN*, selected by the user as the starting point of the transformation process. The algorithm is iterative and is guided by two data structures, WORKLIST, containing all the nodes of the original AST that need to be processed, and DONE, containing the nodes of the original AST that have already been processed. WORKLIST initially contains only *SN*. Each iteration takes the first AST node off of the work list and searches for a rule whose pattern matches the term rooted at that AST node. If a rule is found, the algorithm stages the transformation described by the rule for later transformation, updates the AST node with a record of the chosen transformation to

Input: R = Rule List, P = Program to transform, SN = Selected Node
Internal Variables: WORKLIST, DONE

Procedure Main
  WorkBlock(SN)
  while WORKLIST $\neq \emptyset$
    $N_w \leftarrow$ WORKLIST.popfirstelement()
    WorkBlock(N_w)
  end while

Procedure WorkBlock(SN)
  (N,R) $\leftarrow$ FindRule(SN)
  DONE $\leftarrow$ DONE $\cup$ {N, SN}
  if R == null
    return
  $N' \leftarrow$ SUBST(N, R)
  if $N' \neq$ FAIL
    RecordInAST(N, N')
    D $\leftarrow$ Get the dependencies of N
    for $d \in$ D where $d \notin$ WORKLIST $\wedge$ $d \notin$ DONE **do**
      WORKLIST $\leftarrow$ WORKLIST.appendtoend(d)
    end for
  else if User choice is to roll back $\vee$ some tag value is $\perp$
    Roll code back to original state
    Abort transformation
  else
    Integrate the chosen tag values in the transformed code
  end if

Figure 5. Outline of the transformation algorithm

be performed at the end of the rule application process, and updates the end of the work list with the dependencies of the original term rooted at the AST node. Whether or not a rule is applied, the algorithm adds the AST node to DONE. Note that the algorithm does all rule matching before any transformation. Concretely, it collects the transformations that need to be performed on each node and after all of the nodes have been processed, *i.e.*, the work list is empty, it performs all the transformations at once. In case of conflicting transformations where multiple transformations are staged on the same tree, the transformations that were staged later overwrite the previously staged transformations. Our implementation logs the situations where such overwrites have taken place and makes the log available for inspection by the user. The rest of this section describes the key procedures of the algorithm.

4.2.1 Finding a Node at Which to Apply a Rule

The function FINDRULE(*SN*) searches top to bottom through the rule list to find a rule whose pattern matches an AST that contains the node *SN*. Specifically, *SN* must be matched either to part of the concrete syntax of the rule pattern or to a metavariable. It cannot be a proper subtree of the AST matched to a metavariable. The motivation behind this matching strategy is that the algorithm may need to search up AST ancestors from *SN* to find a node where a rule can be applied, but the search should be limited to subtrees that are dependent on *SN*. The result is a pair of the root of the matched AST, N, and the chosen rule, R, or '(*SN*, null)' if no rule is found.

As an example of the matching process, consider the AST `e(f(g(h(x))))`. In this AST, if the node *SN* under consideration is rooted: **i**, at `f`, **ii**, at the call to `f`, **iii**, at `g`, **iv**, at the call to `g`, or **v**, at `h(x)`, then the matching of the pattern `f(g($i$))` will succeed. In all of these cases, FINDRULE returns the node at the call to `f`. The pattern does not match if *SN* is rooted at `x`, because in that case *SN* is merely a proper subterm of the term `h(x)` matched by `$i$`.

4.2.2 Substituting the Right-Hand-Side of a Rule

The function SUBST(N, R) performs the following operations on node N and rule R:

1. Unify R's *pattern* with N and substitute all metavariables in R's *transformation* accordingly. The transformation may still contain *scope applications*, *tag updates*, and *tag references*.
2. Process all scope applications $scope(\$\text{S}\$)$ left to right, in depth-first pre-order by applying a scoped rule $scope$ to the node n_s as described in Section 4.2.3, where n_s is the AST node matched by S.
3. Replace each tag reference with the corresponding tag value. Tag references have the form $scope.t$ for scoped rule $scope$ and tag t. Each scoped rule with such a tag reference may be applied at most once per *transformation*, and thus the notation is unambiguous.

   If the value of any tag $scope.t$ that occurs in the *transformation* is $\top$, *i.e.*, inconsistent, then ask the user whether to roll back the entire transformation. If the user chooses roll back, then abort and return FAIL. If the user chooses to continue with the transformation, and the tag value is $\top$, ask the user to choose one of the proposed tag values. If the tag value is $\bot$, then the transformation has to be rolled back without user choice. **WorkBlock** can fail only when the tag value is $\bot$.
4. Use the function **RecordInAST** function to append the returned value to the end of list of the transformations to be performed.

4.2.3 Applying a Scoped Rule

A scoped rule is applied to the nodes of the AST rooted at node n_s (referred to as *scoped node*) as follows. The process of searching for nodes to which to apply the rules in a scoped rule is done in depth-first pre-order, and is thus quite different from the process for searching for nodes to which to apply a top-level rule, based on dependencies. Indeed, the process of applying a scoped rule is simply an extension of the dependency-triggered process of applying the top-level rule that invokes it, and thus it effects the complete processing of the scoped nodes.

Our algorithm first initializes to $\bot$ any tag in the scoped rule that is not explicitly initialized. It then traverses all of the scoped nodes in depth-first pre-order and, for each node N, applies the first rule within the scoped rule, if any, whose pattern matches the subtree rooted at N, according to the process described in Section 4.2.1. This traversal ignores the set DONE of previously transformed nodes, neither checking nor recording scoped nodes. However, for any scoped node that the algorithm transforms, it adds all nodes connected via dependency links that are not in DONE to the WORKLIST.

Our use of exhaustive strategy for applying a scoped rule to all scoped nodes allows tags to implement $\forall$ quantification. While updating a tag, if the current value is not $\bot$, a scoped rule ensures that the new value for the tag is equivalent to the current value. Otherwise, the tag is set to $\top$ to indicate disagreement.

By not consulting or extending DONE, the exhaustive strategy allows a given node to be transformed by any number of scoped rules in addition to at most one top-level rule. Whenever the algorithm observes multiple proposed transformations for an AST node, it applies the transformation that was recorded last. However, the system records the discarded earlier transformations in a log file and displays them to the user.

We summarize the differences between rule application for the top-level rules and the scoped rules as follows:

- The rules in a scoped rule are applied to all the scoped nodes, using a depth-first pre-order traversal. Top-level rules are applied to nodes chosen in the order in which the WORKLIST was updated via dependencies.
- Scoped rules do not update DONE, whereas top-level rules always update DONE.
- Top-level rules do not trigger user interaction. Scoped rules trigger user interaction when there is a failure to arrive at a unique tag value. The user can choose to roll back or select one of the observed alternative options for the tag value.

4.2.4 Dependencies

Our algorithm gathers the dependencies of a node N by following its dependency links (Section 4.1), whenever those are part of subterms that are not excluded via **notransform**. The algorithm first traverses the node N in depth-first pre-order and for every encountered node during this traversal, the algorithm process the dependency links in the order determined by their link category (as in Section 4.1).

Automatic Renaming A transformation rule that renames a declared variable or a field declared as part of a type definition also implicitly renames that variable or field consistently throughout the variable's scope. We perform such renaming after applying all top-level rules.

4.2.5 User Interactivity

Our approach is interactive and therefore contains 'choice points' where the user needs to interact with our system to aid the migration process. The user interacts with our system to make the following choices:

- The user chooses the starting point as a code segment.
- If a tag update fails when applying a scope rule to a set of nodes, our system presents the user with the choice of either rolling back the transformation or selecting from the list of values that were proposed for that tag.

4.2.6 Formal Properties

In this section, we discuss the formal properties of our algorithm: termination, confluence, and determinism.

Termination. Our algorithm only processes nodes from the original AST, and processes each node by a top-level rule at most once. As the original AST is finite, the number of iterations of the while loop in the algorithm is finite. Each node contains a finite number of dependency links that can be added to the worklist and a node is never added to the worklist more than once. Each iteration processes at most a set of nodes from a scope and there is a finite number of scope calls per iteration; the number of such nodes is also bounded by the number of nodes in the original AST. Thus, termination is guaranteed.

Confluence. The result of our approach depends on the starting point chosen. For example, if the user transforms the term `a + b` with one rule replacing `a` by `1` and with another rule replacing `b` by `2`, the result will be `1 + b` if the user selects `a` as the starting point, and `a + 2` if the user selects `b`. In both cases, the dependency relations do not trigger transformation of the other argument because no transformation rule applies to the addition expression itself.

Determinism. Overall, the approach is deterministic, in how nodes are added to the worklist (depth-first pre-order traversal), how they are removed from the worklist (first-in-first-out), and how rules are selected. Nevertheless, the order in which nodes are added to the worklist, specifically the processing of the subterms, depends on the structure of the AST, which may not be known by the user. All of the matching for rule selection done by our

approach depends only on the structure of the original program's dataflow graph, and thus it makes no difference to the rule matching whether a particular node is treated earlier or later in the transformation process. The order in which nodes are treated does, however affect which transformation is chosen when multiple transformations accumulate at a single node, and thus in this case the user may not automatically obtain the expected result. We record such conflicts and emit suitable warnings to allow the user to manually intervene later.

Complexity. Our algorithm in the worst case visits each node once, and for each node applications of scoped rules can process all the nodes rooted at the scoped node once. The worst case complexity of our algorithm is therefore $O(n^2)$ where n is the number of nodes in the AST representing the program.

4.3 Limitations

Our algorithm does not perform transformations in-place; instead, it collects transformations at AST nodes, and applies them after all rule matching has taken place. This limitation implies that the application of one rule cannot be sensitive to the effect of applying another rule and that multiple transformations may accumulate for a single node, all but one of which will be thrown away. This limitation stems from the implementation framework that we use, Eclipse CDT, which does not allow in-place transformations. We log the scenarios where multiple transformations are staged on a single node.

Another limitation of our approach is expressiveness. Our language allows matching a code fragment with metavariables and transforming it into another code fragment that makes use of the code fragments bound to the metavariables. Our approach depends on the existence of a one-to-one mapping from a code fragment in the existing data representation to a corresponding code fragment in the target data representation. For example, in order to express vectorizing only alternate elements of an array, our language would require special load and store functions from the Vc library that provide this functionality. Thus, expressing such migrations using our approach would require support from the library supporting the target data representation.

5. Case Studies

We have evaluated our algorithm through three case studies. In each case study, we start with an input specification, an input program and a starting node and apply our tool **DMF** (Data migration framework), based on our algorithm, over it. We run through each step of the algorithm and give a glimpse of the evolution of the work list and the rules being applied.

5.1 Vectorization

Our first case study is based on a tutorial presented on the Vc website [10] that details the conversion from a scalar program to a vectorized program. As previously discussed, the Vc API allows explicit vectorization of C++ source code. We started by writing a specification that describes how to transform scalar code into Vc-style vector code (**Figure 6**). The specification has three main top level rules:

- The rule in **Block A** transforms an array of structures that contain either int or float fields into an array of structures of corresponding vectorized int_v or float_v fields. The constraints on the fields are checked by the scoped rule **vc_struct** in **Block A'**.
- The rule in **Block B** performs vectorization on **for** statements, reducing their number of iterations according to the size of the machine vector registers. The scoped rule **vc_for** in **Block B'** performs most of the work of this transformation.

```
// Block A:
// top-level rule to transform an array of structs
decldefinition struct $structname$ {$body$}[$s$]; ==>
  struct $structname$v
  { vc_struct ($body$) }[$s$/ vc_struct.vctype::size ];
// Block A': scoped rule to transform the body of a struct
// definition and compute the tag vctype within the body
scope vc_struct {{
        tag vctype;
        declaration int $a$; ==> int_v $a$v;
          {{ updatetag vctype := int_v }}
        declaration float $a$; ==> float_v $a$v;
          {{ updatetag vctype := float_v }}
        declaration $T$ $a$; ==> $T$ $a$;
          {{ updatetag vctype := top }}
}}
// Block B: top-level rule to perform
// vectorization on for statements
statement for(int $i$ = 0; $i$ < $limit$; $i$++){$body$}
  ==> for(int $i$ = 0; $i$ < $limit$
  /vc_for.vctype::size; $i$++){ vc_for($body$, $i$) }
// Block B': scoped rule to transform the body of a for
// statement and compute the tag vctype on the loop body
scope vc_for ( $ind$ ) {{
  tag vctype;
  tag loopindex = $ind$;
  expression $a$[$i$].$b$ ==> $a$[$i$].$b$
    {{ updatetag vctype :=
         ( int_v : decltype ($a$[$i$].$b$) == int ,
           float_v : decltype ($a$[$i$].$b$) == float ,
           top : otherwise );
       updatetag loopindex := $i$}}
  expression std::sqrt($expr$) ==> vc::sqrt($expr$)
  expression std::atan2($expr$) ==> vc::atan2($expr$)
  declaration int $a$; ==> int_v $a$v;
       {{ updatetag vctype := int_v }}
  declaration float $a$; ==> float_v $a$v;
       {{ updatetag vctype := float_v }}
  // Rule CE
  statement  if ($c$) { $x$ += $e$; } ==> $x$($c$) += $e$;
}}
// Guidance rule
expression $a$[$i$] ==> $a$[$i$]
```

Figure 6. Case Study Vc Specification

- The guidance rule, at the end of the specification, serves to trigger the propagation of the array index variable to the work list, the need for which will be illustrated in the algorithm walk through below.

The above rules assume that the size of the array is evenly divisible by **float_v::size** or **int_v::size**. To make our rules more general, we can force them to compute the array size rounding up; we omitted this step for readability.

We use the specification shown in **Figure 6** to transform the code shown in **Figure 1**. As a selected node, we choose the declaration of the **input** array, **CartesianCoordinate input[1000]**, and (implicitly) the corresponding type definition struct CartesianCoordinate. The rule in **Block A** in **Figure 6** then applies to this **decldefinition** node, via the following steps:

- **DMF** applies the rules of the scoped rule **vc_struct** (**Block A'**) to the body of the structure definition. This scoped rule contains three rules, for int, float, and other-typed fields, respectively. The first two rules transform the type to the corresponding vectorized type and update the tag with the new type name. The third rule updates the tag with $\top$, which will trigger a failure if **DMF** attempts to apply the rule.
- Reflecting the transformation of the structure definition back to the calling context, the rule in **Block A** uses the vectorized type

name collected in **vc_struct** to transform the size of the array **input**.

Once the declaration of the array **input** and the definition of the structure type CartesianCoordinate are transformed, all the other uses of the identifier **input** and the type CartesianCoordinate are put into the work list; this includes the reference to **input**, specifically its use as part of the array subscript expression **input[i]** for which the guidance rule will apply. The guidance rule does not make any transformations but rather guides the rule application to the declaration of **i** from the **for** statement's initializer clause as part of the the **for** statement itself, which would be transformed.

The **for** statement is then transformed using the rule in **Block B**. The steps in this transformation are similar to those used to transform the structure.

- The **DMF** tool applies the scoped rule **vc_for**, in **Block B'**, to the body of the **for** statement in order to ensure that it is vectorizable. A vectorizable **for** statement is one whose iterations are independent of each other. In particular, we must ensure that inside such a **for** statement, all the array-indexed references of the same vectorizable type. We also require that independent iterations have array indexed references with the same index as the iterator of the **for** loop itself. This is a sufficient condition and not a necessary condition for independent iterations.
  - **DMF** then uses the array subscript expression rule and the declaration rules to update the tag **vctype**. The use of this tag ensures that all references inside the array are of the same type, which must be either float or int.
  - **DMF** uses the tag **loopindex** to ensure that all the array indexed elements share the same index. The **loopindex** tag is initialized to the value of the parameter ind (which is set to the iterator of the **for** statement). This ensures that **loopindex** is updated only if the index i of every array indexed element is the same as ind. Otherwise the value of **loopindex** would conflict and result in a tag update failure. Our specification is not exhaustive in its characterization of when loop iterations are independent but is sufficient for our example.
  - The **Rule CE** in **Block B'** implements the vectorization of conditional increments. Other conditional updates can be expressed analogously.
- **Block B'** also transforms known scalar operations into corresponding Vc operations.

Finally, **DMF** does automatic renaming (as described in Section 4.2.4) on all of the uses of **input** and **output**. The result is the code shown on the right side of **Figure 1**.

5.1.1 Guidance Rule

A guidance rule is a regular transformation rule that expresses a transformation that is only semantic, not syntactic. In this example, the guidance rule expresses that the array changes from scalar to vectorized. While this change requires no syntactic modifications, it does affect dependencies. Specifically, in the transformation in Figure 1, the algorithm gets **output[i]** from the worklist and examines it. The guidance rule identifies **i** as a dependency, and thereby leads our algorithm to the **for** statement, where it can perform the rest of the migration. Without the guidance rule, our algorithm would not understand that the migration needs more work at this point and would miss the need to propagate the dependencies of **i**.

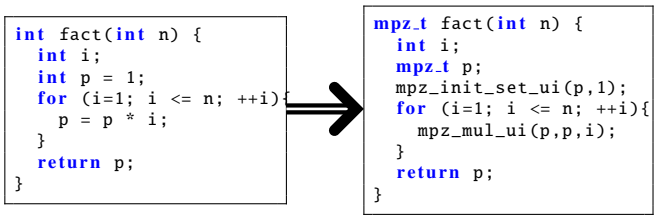

Figure 7. Integer to big integer conversion using GMP

```
statement int $p$ = $c$; ==>
  mpz_t $p$; mpz_init_set_ui($p$,$c$);
  notransform : $c$
expression $a$ = $b$ * $c$ ==> mpz_mul_si ($a$, $b$, $c$)
  notransform: $c$
  where decltype($c$) == int
expression $a$ = $b$ * $c$ ==> mpz_mul_ui ($a$, $b$, $c$)
  notransform: $c$
  where decltype($c$) == unsigned int
expression printf("%d", $p$) ==>
  mpz__out_str(stdout, 10, $p$)
```

Figure 8. Case Study GMP Specification

5.2 GMP Specification

The second use case is based on an example taken out of a GMP tutorial [20]. The GNU Multiple Precision (GMP) Arithmetic Library provides arbitrary precision arithmetic for C and C++ programs [9]. To illustrate its use, we converted an integer factorial function to work on multi-precision values and to return a multi-precision result. **Figure 7** shows the original code (left) and the conversion result (right), from the tutorial. **Figure 8** shows part of a specification that implements this transformation.

To initiate the transformation on the code shown on the left side of **Figure 7**, we select the declaration and initialization of the local variable **p**. The first rule can be applied to the initialization statement. If the declaration and initialization were two separate statements, then the transformation specification would have to be extended with two separate rules. The set of dependencies of the initialization statement includes the references to **p** in the assignment statement in the body of the **for** loop. There is a rule to transform this assignment of a multiplication (involving **unsigned int**) to a GMP library call. At this point, **DMF** would not put the uses of the variable **i** in the worklist, due to the **notransform** annotation. Finally, **DMF** traces the dependency flow into the final print (not shown) of the computed factorial and updates it with the corresponding GMP operation for printing numbers.

5.3 Array of Struct vs Struct of Arrays

We used **DMF** to perform the transformations shown in a tutorial published by Intel [1] that illustrates a useful data representation migration: transforming data organized as an Array of Structures (AoS) into a Structure of Arrays (SoA). This transformation allows the compiler to access data more efficiently in many applications, by improving locality. Such a transformation also helps the compiler vectorize loops that iterate over the array. The example consists of some variable declarations and an 8-line loop. The program uses array notations that are not native to C++, but are from the Cilk extension, as discussed on Intel's Cilk website.[1] In order to process

[1] https://www.cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_1.2.htm

```
struct node1 {
  float x, y, z;
};
struct node1 NODES1[1024];

void main() {
  float dist[1024];

  for (int i = 0; i < 1024; i += 16) {
    float x[16], y[16], z[16], d[16];
    for (int i1 = 0; i1 < 16; i1++) {
    x[i1] = NODES1[i + i1].x;
  }
  for (int i1 = 0; i1 < 16; i1++) {
    y[i1] = NODES1[i + i1].y;
  }
  for (int i1 = 0; i1 < 16; i1++) {
    z[i1] = NODES1[i + i1].z;
  }
  for (int i1 = 0; i1 < 16; i1++) {
    d[i1] = sqrtf(x[i1] * x[i1]
    + y[i1] * y[i1] + z[i1] * z[i1]);
  }
  for (int i1 = 0; i1 < 16; i1++) {
    dist[i + i1] = d[i1];
  } } }
```

```
struct node1 {
  float x[1024], y[1024], z[1024];
};
struct node1 NODES1;

void main() {
  float dist[1024];
  for (int i = 0; i < 1024; i += 16) {
    float x[16], y[16], z[16], d[16];
  for (int i1 = 0; i1 < 16; i1++) {
    x[i1] = NODES1.x[i + i1];
  }
  for (int i1 = 0; i1 < 16; i1++) {
    y[i1] = NODES1.y[i + i1];
  }
  for (int i1 = 0; i1 < 16; i1++) {
    z[i1] = NODES1.z[i + i1];
  }
  for (int i1 = 0; i1 < 16; i1++) {
    d[i1] = sqrtf(x[i1] * x[i1]
    + y[i1] * y[i1] + z[i1] * z[i1]);
  }
  for (int i1 = 0; i1 < 16; i1++) {
    dist[i + i1] = d[i1];
  } } }
```

Figure 9. Going from Array of Struct to Struct of Array

```
decldefinition
 struct $structname$ { $structbody$ }
 $structobj$[$limit$]; ==> struct $structname$
  { struct_scope($structbody$, $limit$) }
  $structobj$;
scope struct_scope($l$){{
 declaration $type$ $fieldname$ ==>$type$ $fieldname$[$l$]
}}

expression $name$[$i$].$field$ ==> $name$.$field$[$i$]
```

Figure 10. AoS to SoA Specification

the Cilk code, we wrote C code performing the same computations. It is to be noted that Cilk code is designed for parallelism and our C code will not run in parallel.

The specification for converting a structure of Arrays into an array of structures, as shown in Figure 10, again illustrates the usefulness of scoped rules with parameters. The first rule applies a scoped rule, **struct_scope** to the body of the struct definition. The scoped rule transforms every member field inside the struct definition into an array declaration with the limit set to **l**, which is a parameter set to the number of elements in the declared array. In our code this parameter would be **1024**. There is then one more top-level rule that modifies the array references.

6. Implementation and Results

We have prototyped **DMF** in Java (∼1200 LOC) using a parser that acts as a front-end for our language and Eclipse CDT [7] as the basis of our transformation system. We ran our examples on an Intel Core i7-4770 CPU at 3.40GHz with 2×4 cores, running a 64-bit Ubuntu. For the Vc examples, we used the gcc compiler (v4.9.4) settings specified by the Vc make-files. In this section, we report on our experiments to validate that each of our transformed programs has met the intended purpose of the transformation.

6.1 Vc

The primary purpose of the Vc API is to improve performance. We measured the running time of the scalar version of the loop in our non-Vc code from Figure 1. We compared it to the running time of the loop in the Vc code on various architectures (AVX, AVX2, SSE), all of which are x86 extensions that support SIMD (single instruction, multiple data) instructions. For each of our settings we ran the existing **for** loop 100 times by wrapping it within another for loop that runs for 100 iterations. We removed the first run to account for warm-up effects, without restarting the program. The C++ compiler optimization levels were set to **O3** for all of the runs. **O3** instructs the gcc compiler to auto-vectorize as much as possible, thereby eliminating the possibility that the runtime measurements we get are something that the compiler could have achieved automatically. **Figure 11** summarizes the results. The x-axis shows the duration in microseconds, and the y-axis shows the name of the tested architecture and whether it benchmarked the post-transformation Vc version (vector) or the original unvectorized version (scalar). On each of the vector architectures, AVX, AVX2, and SSE, the non-Vc version of the loops takes more than twice as long as the corresponding vector version.

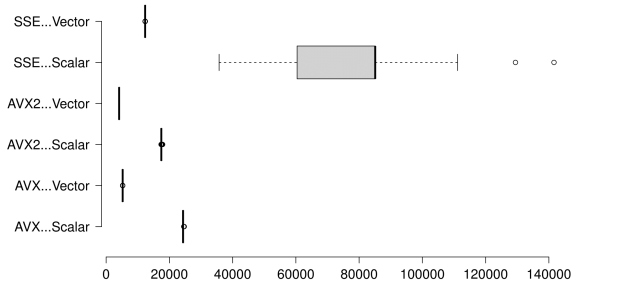

Figure 11. Vc runtimes, as box plots summarizing 99 runs.

6.2 GMP

The primary purpose of GMP is to represent numbers of unbounded size and precision. In our tests, for example, the int-based implementation of factorial prints 2004310016 for the factorial of 15, indicating an integer overflow, while the generated GMP implementation prints the correct result, 1307674368000.

6.3 Array of Struct vs Struct of Array

Choosing between the Array of Structures and Structure of Arrays representations is a known challenge in practice [1, 22], with either representation potentially advantageous, depending on usage.

The benefits of the array of structures to structure of arrays transformation and the results are discussed in detail at the website [1]. We calculated the running time of both the Array of Structures and the Structure of arrays versions of the program in Figure 9 by running each version 300 times and calculating the average time per run. The array of structures runs took on average about twice as long as the structure of arrays runs.

7. Further Evaluation

On top of our case studies, we also performed a few experiments with the GMP specification to illustrate the scalability and generality of our transformation approach. Scalability validates if our approach scales to large applications and generality tests whether specifications written in our language are applicable to more than one software project. We first describe the running time of our search algorithm on a larger code base and set of specifications. We then describe our experiences in applying specifications created for one program to different, randomly selected programs, and testing the transformed results.

7.1 Scalability

To evaluate the scalability of our approach, we selected a nontrivial piece of software to which one of our data representation migrations is pervasively applicable. As a transformation, we selected our integer to big integer conversion, since it is the most broadly applicable of our examples (requiring only the use of int values for arithmetic). As a target program, we selected a nontrivial piece of software that already uses the GMP library, and manually converted it to use integers, thus providing both the source and the desired target for our transformation process.

We selected the software by searching GitHub[2] for a project that contains `include<gmp.h>` with a source code size of at least 10,000 bytes. The first result[3] was a file of 3616 lines of code that makes 1167 references to the GMP API, involving variable declarations and function calls. The code is fairly complex, defining functions to add, multiply and do other arithmetic operations on big polynomials. We replaced all calls to GMP operations with integer arithmetic operations and verified that the resulting code compiled. We used the GMP specification (28 rules), and ran this specification on the resulting integer code and a starting point involving a declaration of a variable of type r_big_polynomial. This starting point was sufficient to reach all the code that used the GMP library in the original implementation. The transformations completed in 133 seconds in the same machine setup used in Section 6.

The resulting code was not exactly the same as the code from which we started. Indeed, there are multiple ways to transform some kinds of integer operations. In particular, an assignment of the form a = b can be translated both to mpz_init_set(a,b) and mpz_set(a,b). The difference between the two functions is that the former should be used when initializing a variable for the first time, while the latter is intended to be used for a subsequent update. **DMF** currently does not support detecting such differences automatically. There were 35 such differences, all of them assignment related, between the generated code and the original GMP implementation from GitHub. **DMF** logged the available options in cases where multiple transformations are possible and in all of the cases, we observed that the expected transformation was among the logged options. This result indicates that the approach can complete the search of a larger code with a realistic specification in a couple of minutes.

7.2 Generality

In order to illustrate the generality of our specifications, we applied our GMP specification to arithmetic algorithms in three algorithm libraries. We randomly selected these libraries by providing three search terms and picking the top repository obtained for each term. The search terms were:

- **arithmetic algorithms** with language set to C, which returned the repository `https://github.com/celeritas17/arithmetic_algorithms`.
- **algorithms** with language set to C++, which returned the repository `https://github.com/PetarV-/Algorithms`
- **algorithms** with language set to C, which returned the repository `https://github.com/davidreynolds/algorithms`

We randomly picked from these libraries six algorithms that run on integers and applied our specifications to transform their code to run on big integers. Unlike the previous experiment, we had no ground truth to compare to. The selected code and the results are summarized in Figure 12.

[2] `https://github.com`

[3] `https://github.com/PuRoTeam/PuRoAtkinMorain/blob/7620e46a59fb0e8659bfbed7ee75ad3af5de35f5/tools/polynomial.c`

On manual inspection, we found that **DMF** performed most of the required transformations correctly, but we did encounter a few issues. One of the examples had a condition **if ((first+second) == N)** where all of **first**, **second** and **N** had to be transformed from integer to big integer. Unfortunately, the GMP library does not provide a function that adds two big integers and returns a big integer. It merely has a function **mpz_add** which takes three parameters, where the sum of the last two is stored in the first parameter. We thus had to modify the input code to store the result in a temporary variable before the check. Another challenge, which was recurring, was the commutativity of arithmetic operations. Our specification contains rules for transforming expressions of the form **a op const** into **mpz_op_ui(a, const)** where **const** is a constant value. However, some of the expressions had the form **const op a**. We thus had to create rules for both cases, which mapped to the same GMP call. The rules differentiate the position of the constant using a **where** clause. This indicates that specifications written using our approach are generalizable to not just one project but to multiple code bases requiring the same kind of transformation. Crucially, we did not have to remove any rules, suggesting that developers can share and grow our specifications to express common transformation tasks.

8. Related Work

We begin this section by discussing features in other transformation systems. We then discuss other related work.

8.1 Related Features

Among existing program transformation systems for C/C++, the two most closely related systems are Coccinelle and Stratego.

8.1.1 Coccinelle

Coccinelle [17] is a program matching and transformation engine that provides SmPL, the Semantic Patch Language, for specifying desired matches and transformations in C code. Syntactically, Coccinelle supports a patch-like notation for describing pattern matching and transformation rules. A simple example is:

```
@@
expression e1;
@@
- fn1(e1)
+ fn2(e1, 20)
```

The above specification declares a metavariable **e1**, matches the pattern **fn1(e1)** and transforms it to **fn2(e1, 20)**. This specification would e.g. transform the code fragment **fn1(x)** into **fn2(x, 20)**.

Coccinelle supports well finding the existence of a match of a pattern, but is less well suited to checking that all instances of a pattern satisfy certain properties, as provided by our tag construct. As an example, the following specification re-implements using Coccinelle our transformation of a structure containing fields that all have the same type to its vectorized counterpart:

```
1  /* get the type of the first field of each structure */
2  @r@
3  identifier i,varname;
4  type T;
5  @@
6  struct i { T varname; ... };
7
8  /* Mark other fields having the same type */
9  @others@
10 identifier r.i,varname;
11 type r.T;
12 position p;
13 @@
14 struct i { ...  T varname@p;  ... };
15
16 /* Find fields having a different type */
17 @bad@
```

Algorithm	Description	Lines/Sites	Code (Before/after)
Insertion Sort	Standard sort algorithm.	17/19/4	github.com/PetarV-/Algorithms/blob/b5ed8b5/Sorting%20Algorithms/Insertion%20Sort.cpp github.com/krishnanm86/Pepm2017Eval/blob/6fd7897/insertionsort.c
Subset Sum	Find integer pairs in an integer array that they sum up to a given number N.	27/33/11	github.com/davidreynolds/algorithms/blob/7d59299/c/subset_sum/int_pair_sum.c github.com/krishnanm86/Pepm2017Eval/blob/6fd7897/intpairsum.c
Binary Search	Standard search algorithm.	16/16/4	github.com/davidreynolds/algorithms/blob/7d59299/c/searching/binary_search.c github.com/krishnanm86/Pepm2017Eval/blob/6fd7897/binarysearch.c
Divide and conquer	Divide-and-conquer search for largest array element	17/19/3	github.com/davidreynolds/algorithms/blob/7d59299/c/searching/divide_and_conquer.c github.com/krishnanm86/Pepm2017Eval/blob/6fd7897/divide_and_conquer.c
Fast Multiply	An interesting method to multiply integers.	18/28/13	github.com/celeritas17/arithmetic_algorithms/blob/5d79830/src/div_and_mult.h github.com/krishnanm86/Pepm2017Eval/blob/6fd7897/multiplynew.c
Euclid GCD	Euclid's algorithm for finding the greatest common divisor.	14/19/6	github.com/celeritas17/arithmetic_algorithms/blob/5d79830/src/div_and_mult.h github.com/krishnanm86/Pepm2017Eval/blob/6fd7897/euclid_gcdnew.c

Figure 12. Results of generality study of GMP specifications. Lines/Sites list lines before/lines after/transformed sites.

```
18  identifier r.i,x;
19  type T;
20  position p != others.p;
21  @@
22  struct i { ... T x@p; ... };
23
24  /* Make new type name if rule 'bad' was not satisfied */
25  @script:ocaml change depends on !bad@
26  _i << r.i;
27  t << r.T;
28  t1;
29  @@
30  t1 := Coccilib.make_type (t^"_v")
31
32  /* change the types to use the new name */
33  @@
34  identifier r.i,x;
35  type r.T, change.t1;
36  @@
37  struct i {
38     ...
39  - T
40  + t1
41     x;
42     ...
43  };
```

To check that all of the fields have the same type, the first rule (lines 1–6) obtains the type of the first field. The second rule (lines 8–14) marks all of the fields that have the same type as that one. The third rule (lines 16–22) identifies whether there are fields that have another type, the fourth rule (lines 24–30) makes a new type name if it has been found that all fields have the same type, and the fifth rule (lines 32–43) makes the transformation.

Thus, while it is possible to express $\forall$ quantification using Coccinelle, the implementation is much more complex than the dedicated abstraction provided by our language. Furthermore, rules in Coccinelle are triggered by syntactic pattern matching, that can be less precise than the dependency based rule triggering provided by our approach. For example, the various rules assume that all of the declarations of a structure type `i`, where `i` is a metavariable established in the first rule, are one and the same. This hypothesis could be incorrect if the code contains multiple definitions due to `#ifdefs` (Coccinelle does not first apply the C preprocessor). Our approach triggers rule applications to specific AST nodes, eliminating this ambiguity.

8.1.2 Stratego

Stratego is a language independent code transformation system [3]. Internally, terms are represented as abstract syntax trees, such as **Call("double", [Minus(Var("y"), Int(10))])** represents the expression double(y - 10), but front ends are provided for a variety of languages allowing the use of concrete syntax. Stratego, like our approach, permits application of rules in a guided fashion. For this purpose, Stratego supports the definition of *strategies*, which allow rewrite rules to apply conditionally and in specific user-defined orders. For example, the strategy $r1 < r2 + r3$ involving the rules `r1`, `r2`, and `r3` indicates that rules `r1`, `r2`, and `r3` should be applied as follows. Begin by applying `r1`. Upon successful application of `r1`, apply `r2` to the result; if `r1` fails, then apply `r3` to the original term. if `r2` fails, then do not backtrack. Stratego, however, has no built-in mechanism for triggering rule application from dependency links.

8.1.3 IDE Refactorings

Refactoring environments like Eclipse [6] and Netbeans,[4] provide simple code transformations like 'rename variable', which are supported by data-driven transformations. For example, renaming a variable in one location also transforms all of its uses and declarations elsewhere. However, refactoring systems offer only a fixed set of transformations that are difficult to extend or customize.

8.2 Abstract Datatypes

Abstract datatypes [14] (ADTs) can render program transformation unnecessary: once implemented, ADTs allow software developers to swap functionality by changing a single line. However, ADTs require explicit preparation and abstraction, in the form of class or module interfaces, they may come at a performance cost (e.g., due to dynamic dispatch), and they may be insufficiently expressive: we are unaware of mainstream programming languages that can use ADTs to flip between array-of-structs and struct-of-arrays-style data representation for arbitrary user-defined data structures.

8.3 Other Related Work

One of the first works on program transformation languages was that of Burstall and Darlington [4], who described a theoretical framework for describing program transformation systems. They primarily targeted the problem of program derivation, rather than data representation migration issues.

[4] `http://wiki.netbeans.org/Refactoring`

Our work is closely related to term rewriting tools. Tom [16] is a general purpose tool for extending existing programming languages with pattern matching. It is very lightweight and can be used to write simple term rewriting extensions for a variety of imperative languages. Tom, however, does not provide abstractions for managing data propagation information and dependencies, and thus falls short when addressing the issues of expressing rewriting rules for data representation migrations. Batory et al. [2] proposed an approach for data structures in the form of minimal libraries that contain primitive building blocks and generators. Although such an approach can be used to build ADTs that can have many implementations and these implementations ease the migration between the implemented data types, our approach supports the problem of data migration and the challenges associated with it directly.

Our work also involves program analysis, through our tags. Although program analysis has been used to address issues related to performance and security, few works incorporate program analysis along with program transformation. Spoon [18] is a library for implementing analyses and transformations of Java source code. Although Spoon provides features that can be used to combine program analysis with program transformation, the developer has to write their own transformation and analysis code using a CDT-like low-level library by hand. Our approach on the other hand simplifies the task of writing transformation rules for migrating data representations by providing simple forms of program analysis and information propagation, via scopes and tags.

Our work is a form of incremental source code refactoring, by incrementally selecting transformations to be applied. Reichenbach et al. [19] also present an incremental transformation approach, with behavioral guarantees ensured by a general comparison mechanism, albeit for traditional refactorings and with very limited support for automation. Schäfer et al. [21] analogously present a general approach to implementing software refactorings by viewing them as invariant-preserving transformations on programs in an enriched language. Although both approaches use the idea of incremental refactoring to aid development of complex refactoring systems, they fail to address the specific challenges of data representation migration, *i.e.*, the ability to express information propagation across code fragments and flag dependencies for further changes.

In cases where dataflow behaves analogously to types (i.e., when flow sensitivity makes no difference), our work is comparable to work on automatically fixing type errors, analogously to an approach proposed by McAdam [15], who uses a bottom up approach based on a construct called type isomorphisms to suggest methods to repair programs with type errors.

9. Conclusion

In this paper, we have analyzed real world data representation migrations and identified core features that an automated framework must support to perform such transformations. We have designed a language around these features and implemented a prototype tool supporting this language. We have evaluated our language on data representation migration examples chosen from external websites. We have also illustrated the scalability of our algorithm by applying a large specification to a larger code base and have established the difference between our approach and other program transformation frameworks and tools. We therefore find that our approach is effective at expressing and performing a variety of transformation tasks, and offers a novel view on datatype migration.

References

[1] Amanda S. Memory layout transformations. `https://software.intel.com/en-us/articles/memory-layout-transformations`.

[2] D. S. Batory, V. Singhal, M. Sirkin, and J. Thomas. Scalable software libraries. In *SIGSOFT '93, Proceedings of the First ACM SIGSOFT Symposium on Foundations of Software Engineering, Los Angeles, California, USA, December 7-10, 1993*, pages 191–199, 1993.

[3] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008. Special Issue on Second issue of experimental software and toolkits (EST).

[4] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, Jan. 1977.

[5] cppreference.com. decltype specifier. `http://en.cppreference.com/w/cpp/language/decltype`.

[6] Eclipse. `http://www.eclipse.org/`.

[7] Eclipse CDT. `http://www.eclipse.org/cdt/`.

[8] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[9] GNU. GMP library. `https://gmplib.org/`.

[10] M. Kretz. Vc: Cartesian to polar co-ordinates. `http://code.compeng.uni-frankfurt.de/docs/Vc-0.7/ex-polarcoord.html`.

[11] M. Kretz. Efficient use of multi- and many-core systems with vectorization and multithreading. Master's thesis, University of Heidelberg, 2009.

[12] M. Kretz and V. Lindenstruth. Vc: A C++ library for explicit vectorization. *Software: Practice and Experience*, 42(11):1409–1430, 2012.

[13] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[14] B. Liskov and S. Zilles. Programming with abstract data types. In *SIGPLAN Notices*, pages 50–59, 1974.

[15] B. J. McAdam. How to repair type errors automatically. In *Selected Papers from the 3rd Scottish Functional Programming Workshop*, SFP '01, pages 87–98, Exeter, UK, UK, 2001. Intellect Books.

[16] P.-E. Moreau, C. Ringeissen, and M. Vittek. A pattern matching compiler for multiple target languages. In *Proceedings of the 12th International Conference on Compiler Construction*, CC'03, pages 61–76. Springer-Verlag, 2003.

[17] Y. Padioleau, J. L. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys*, pages 247–260, 2008.

[18] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. Spoon: A library for implementing analyses and transformations of Java source code. *Software: Practice and Experience*, 46(9):1155–1179, 2016. spe.2346.

[19] C. Reichenbach, D. Coughlin, and A. Diwan. Program metamorphosis. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, pages 394–418, 2009.

[20] S. Sankaranarayan. Tutorial on GMP. `https://www.cs.colorado.edu/~srirams/courses/csci2824-spr14/gmpTutorial.html`.

[21] M. Schäfer, M. Verbaere, T. Ekman, and O. Moor. Stepping stones over the refactoring Rubicon. In *ECOOP*, pages 369–393. Springer-Verlag, 2009.

[22] K. Sharma, I. Karlin, J. Keasler, J. R. McGraw, and V. Sarkar. Data layout optimization for portable performance. In *European Conference on Parallel Processing*, pages 250–262. Springer Berlin Heidelberg, 2015.

[23] X. Zheng and R. Rugina. Demand-driven alias analysis for c. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 197–208, New York, NY, USA, 2008. ACM.